UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Francisco José de Caldas District University

Kine: Data System Design for a Digital Wallet

Sofia Lozano Martinez

Jose Jesus Cespedes Rivera

Teaching: Carlos Andrés Sierra

Faculty of Engineering
Systems Engineering Curriculum Project
Database II

July 10, 2025

Abstract

This technical report presents the design, implementation, and testing of a hybrid and distributed database architecture for Kine, a digital wallet system inspired by Nequi, developed as an academic project in the fintech domain. The primary objective was to create a robust, scalable, and modular system capable of managing high volumes of transactional operations while storing historical and semi-structured data.

The project began with a business model analysis using the Business Model Canvas, followed by the definition of functional and non-functional requirements, user story formulation, and the development of a conceptual model based on the "10 Steps of Ontology" methodology. This led to the design and implementation of a relational entity-relationship model in PostgreSQL, incorporating database triggers and constraints to enforce business rules and maintain data integrity.

A distributed architecture was integrated by connecting PostgreSQL with MongoDB Atlas, used for storing semi-structured data such as support requests and responses. Although limited by the 512 MB capacity of the free-tier, the integration validated the feasibility of leveraging cloud-based distributed storage for specific data types.

Python scripts were developed to automate the generation of synthetic data, creating thousands of user profiles, account records, and financial transactions to simulate realistic operational scenarios. Tools like Faker, pandas, and random were utilized to introduce variability and realism into the dataset. Performance tests demonstrated the system's ability to handle hundreds of inserts in under one second, confirming its potential to support high-concurrency environments typical of fintech applications.

The system also includes processes for generating user transaction extracts, calculating total incomes and expenses over defined periods, and producing detailed reports. While certain limitations remain—including the absence of user interfaces, advanced security implementations, and large-scale production testing—the project successfully demonstrates key architectural principles and validates the viability of a hybrid relational and NoSQL solution for modern digital banking platforms.

Keywords: database, Nequi, fintech, ontology, relational modeling, digital wallet, PostgreSQL, MongoDB, concurrency, hybrid architecture, synthetic data generation.

Contents

Chapter 1

Introduction

In recent years, the financial technology (fintech) sector has experienced remarkable growth, driven by the need for innovative digital services and the increasing demand for financial inclusion. Digital wallet platforms like Nequi in Colombia exemplify this transformation, offering fully digital services without the need for physical branches. According to Ospina Henao (2025), Nequi processes over 38 million transactions daily and serves more than 21.3 million active users, highlighting the technological complexity and scalability required in modern financial systems Henao (2025).

Motivated by this context, the present project focuses on the development of Kine—a digital wallet system inspired by Nequi. The primary objective was to implement and integrate concepts learned in database systems, transitioning from theoretical knowledge to practical application. Kine aims to replicate core functionalities of digital banking platforms, including user and account management, financial transactions, support operations, and data reporting.

The project began with a comprehensive analysis of Nequi's business model to identify essential components and operational requirements. Using the "10 Steps of Ontology" methodology, a conceptual model was designed and subsequently translated into a physical relational schema implemented in PostgreSQL. The decision to integrate MongoDB Atlas complemented the architecture, allowing for distributed storage of semi-structured data such as support requests and historical records.

A significant part of the project involved generating synthetic data using Python libraries like Faker, pandas, and random to simulate realistic operational scenarios. Performance tests were conducted to evaluate the system's capacity to handle concurrent transactions efficiently, with successful results demonstrating sub-second response times for bulk insert operations.

The structure of this document reflects the complete journey of the Kine project. It includes a review of related literature, the definition of objectives, methodology, system architecture, results obtained, and critical discussions about design decisions and future considerations. Although the system was developed in an academic environment and tested under controlled conditions, it lays a solid foundation for understanding the challenges and solutions involved in designing modern fintech platforms.

This document thus serves both as a technical report and as an academic exercise in applying database principles to the real-world challenges of digital banking systems.

Chapter 2

Literature Review

The evolution of fintech platforms has introduced new challenges for database systems, demanding high transaction throughput, robust security, and continuous availability. Modern financial applications must support millions of operations daily while ensuring strict data integrity and regulatory compliance.

2.1 Relational Databases in Fintech

Relational database systems remain fundamental to financial applications due to their ACID properties and strong transaction guarantees. PostgreSQL, in particular, is widely used for high-traffic systems because of its stability, advanced concurrency management, and native support for features like Multiversion Concurrency Control (MVCC) PostgreSQL Global Development Group (2023). According to Jain (2022), scaling PostgreSQL for high-traffic applications requires careful design of indexes, connection pooling, and partitioning strategies to maintain performance Jain (2022). Benchmarking tools such as `pgbench` further enable performance assessment under simulated loads, demonstrating capabilities of approximately 896.97 transactions per second in default environments with 10 clients PostgreSQL Global Development Group (2024).

These relational technologies are crucial for managing critical transactional data such as user accounts, balances, and financial movements, where consistency and data integrity are paramount.

2.2 NoSQL and Distributed Architectures

As fintech workloads expand, NoSQL databases have emerged as complementary technologies to relational systems, offering flexibility in handling semi-structured and large-scale datasets. MongoDB provides horizontal scalability through sharding, allowing systems to distribute data across multiple nodes to support increasing workloads MongoDB Inc. (2024). The comparison between MongoDB and relational databases emphasizes the trade-off between schema flexibility and transactional consistency MongoDB Inc. (2023).

NoSQL technologies are particularly suitable for handling operational logs, support requests, and historical data where strict relational constraints are less critical. Their document-oriented approach simplifies the storage of heterogeneous data structures and enables rapid evolution of application features.

2.3 Nequi and Regional Fintech Innovations

In the Latin American context, Nequi stands as one of the leading digital wallet platforms. As reported by Ospina Henao (2025), Nequi processes over 38 million transactions daily, serving more than 21.3 million active users Henao (2025). This operational scale demands robust technological infrastructures capable of managing high transaction volumes and ensuring data availability and security.

Nequi's architecture has become a reference model for similar platforms in the region, inspiring projects like Kine to adopt hybrid architectures that combine relational consistency with NoSQL scalability.

2.4 Legacy Systems vs. Modern Architectures

Legacy banking systems are often monolithic and inflexible, relying heavily on rigid relational databases and complex integrations for new services. Modern fintech solutions, however, are moving towards hybrid architectures that separate transactional systems from analytical workloads and adopt microservices for modularity. MongoDB Inc. emphasizes that sharding and distributed deployments are essential to meet modern scalability demands while relational systems like PostgreSQL remain critical for ensuring strong data consistency in core operations MongoDB Inc. (2024); PostgreSQL Global Development Group (2023).

This architectural evolution reflects a shift from purely relational designs to hybrid systems capable of handling both OLTP and OLAP workloads efficiently.

2.5 Implications for Kine

The Kine project has drawn on these industry insights and best practices to design a hybrid architecture that leverages the strengths of both PostgreSQL and MongoDB Atlas. The decision to implement PostgreSQL as the primary transactional engine ensures ACID compliance and high performance for financial operations, while MongoDB Atlas provides a flexible platform for managing semi-structured data like support interactions and archival records.

By aligning with documented benchmarks and architectural guidelines, Kine establishes a solid foundation for scalability and future enhancements, positioning itself as a realistic prototype inspired by leading fintech platforms like Nequi.

Chapter 3

Background

Nequi offers a fully digital banking experience centered on the user, operating without physical branches and relying on a robust technological infrastructure. It is part of the fintech ecosystem in Colombia, with significant partners including Bancolombia, FGA Fondo de Garantías S.A., Visa, Payoneer, and Redeban. Through its mobile application, Nequi enables multiple operations such as account opening, payments, transfers, mobile recharges, cardless withdrawals, and loan applications like Crédito Salvavidas and Propulsor.
This operational model requires a highly scalable and resilient technological platform capable of handling millions of transactions daily. Nequi reportedly processes over 38 million transactions each day—approximately 26,400 transactions per minute—serving more than 21.3 million active users. This scale of operations demands strict requirements in terms of data integrity, security, and system performance.

3.1 Relation to Kine Project

The development of the Kine system was inspired by Nequi's architecture and functionalities. As an academic exercise, the Kine project aimed to replicate essential features of a digital wallet and simulate the technological challenges inherent in a fintech environment.
Key parallels and components implemented in Kine include:

- **User and Account Management:** Kine includes tables and processes for managing user profiles, authentication, device information, and account balances, replicating Nequi's ability to open and manage digital accounts.
- **Transactional Operations:** The system supports the simulation of financial movements such as transfers, bill payments, credit operations, and withdrawals. Synthetic transactions were generated using Python libraries like Faker to mimic real user behaviors and transactional patterns.
- **Support System:** Similar to Nequi's customer support features, Kine integrates a support module where users can create requests and receive responses from agents. This data is stored in MongoDB Atlas to leverage NoSQL flexibility for semi-structured data.
- **Concurrency and Performance:** The project addressed challenges related to concurrent transactions, ensuring data consistency during simultaneous operations—an essential consideration given Nequi's high transaction volumes.

- **Data Extraction and Reporting:** Kine implements functionalities for generating user transaction extracts, summarizing financial activities within selected periods. This reflects Nequi's capabilities in providing users with account statements and transaction histories.
- **Hybrid Database Architecture:** Inspired by the technological diversity required in a fintech environment, Kine adopted a hybrid model combining PostgreSQL for transactional integrity and MongoDB Atlas for distributed storage of support and historical data.
- **Business Rules Enforcement:** Kine incorporated triggers and constraints in PostgreSQL to enforce business rules at the database level, similar to the robust validation and control mechanisms necessary in real fintech applications.

While Kine does not operate at Nequi's production scale or manage real financial data, the project successfully demonstrates the core principles and architectural decisions necessary to build a digital wallet system. The results from testing confirm that the system can handle significant transaction loads, enforce business rules, and manage data distribution between relational and NoSQL databases.
This academic exercise serves as a foundational exploration of how digital financial services can be architected, providing valuable insights into the challenges and solutions that underpin modern fintech applications like Nequi.

Chapter 4

Objetives

4.1 General Objective

To design and implement a hybrid database architecture for Kine, inspired by Nequi, that fulfills identified functional and non-functional requirements, enabling secure, consistent, and efficient operation while supporting scalability and distributed data management.

4.2 Specific Objectives

- Analyze the business model of a digital wallet platform to identify its functional components and operational requirements.
- Define functional and non-functional requirements based on regulatory and business needs specific to the fintech sector.
- Develop user stories to describe the main use cases, ensuring alignment with user expectations and system capabilities.
- Apply the "10 Steps of Ontology" method to generate a conceptual model representing the system's knowledge structure.
- Design and implement a relational entity-relationship model in PostgreSQL, ensuring data integrity, normalization, and efficient query performance.
- Integrate a NoSQL distributed database (MongoDB Atlas) to store semi-structured and historical data, enabling scalability and offloading transactional workloads.
- Develop Python scripts to automate the generation and insertion of test data, simulating realistic operational scenarios for users, accounts, transactions, and support interactions.
- Implement concurrency control mechanisms to ensure data consistency under simultaneous operations.
- Perform performance and load tests to validate system responsiveness and scalability under high transaction volumes.
- Develop processes for generating transaction extracts and reports, providing users with insights into their financial activities.

Chapter 5

Scope

The scope of this project extends beyond the initial analysis and conceptual modeling of the database. It includes the full design and implementation of a hybrid data architecture inspired by the operational model of Nequi, covering both relational and distributed database technologies.

This stage of the project encompasses:

- Analysis of the business model using the Business Model Canvas to identify key functional components.
- Specification of functional and non-functional requirements aligned with the needs of a digital financial service platform.
- Development of user stories to represent core use cases.
- Application of the 10 Steps of Ontology method to build the conceptual data model.
- Design and implementation of a normalized entity-relationship model using PostgreSQL for transactional operations.
- Integration of MongoDB Atlas as a distributed NoSQL solution for storing semi-structured and historical data.
- Creation of data generation scripts using Python (Faker, pandas, random) to populate the system with realistic test data.
- Implementation of business logic through database triggers and constraints to enforce data integrity.
- Execution of performance and concurrency tests to validate the system's responsiveness under load.
- Development of user extract generation processes based on transactional data over defined time ranges.

Although the project includes integration with a distributed system (MongoDB Atlas), this integration was performed using the free-tier service, which imposes a 512 MB storage limit. Therefore, large-scale distributed deployment, real-time analytics, and production-level security measures remain outside the current scope but are identified as potential areas for future development.

Chapter 6

Assumption

Before continuing with the technical design and implementation, it was essential to define a set of assumptions that supported the development of the Kine database model. These assumptions help to delimit the system's operational context and anticipate the conditions necessary for its functionality and testing.

- The processes defined in the business model are assumed to be representative and sufficient for the initial design and implementation phases.
- The platform is assumed to operate under 24/7 availability conditions and be prepared for continuous growth in both user base and transaction volume.
- Users are assumed to primarily interact with the platform via mobile devices, reflecting modern fintech usage patterns.
- All data used for testing and validation is synthetic and generated using libraries such as Faker, pandas, and random, meaning no real customer data is involved.
- The PostgreSQL database is assumed to be capable of handling concurrency through native features like MVCC and transaction isolation levels, without requiring external distributed transaction managers for the current scope.
- The integration with MongoDB Atlas is assumed to be feasible within the constraints of the free-tier plan (512 MB storage), understanding that scaling to larger datasets would require higher-tier subscriptions.
- Security considerations in this project are limited to data consistency and structural integrity at the database level; advanced security measures such as encryption, secure authentication protocols, and production-level compliance are assumed to be out of scope for the current academic implementation.
- The system is assumed to run in a controlled academic environment without external regulatory audits or real-world financial risk.

Chapter 7

Limitation

Although significant progress was achieved in the design and implementation of the Kine system, several limitations were identified that constrain the current scope and indicate areas for future improvement. These limitations stem from technological constraints, the academic context of the project, and the boundaries intentionally established to focus the development effort.

- **Absence of Real System Data:** All data used for testing and validation purposes was synthetic, generated using tools such as Faker and random. While this approach effectively simulates data structure and volumes, it does not perfectly replicate real-world patterns, anomalies, or user behaviors that might occur in a live financial environment.
- **MongoDB Atlas Free-Tier Limitation:** Integration with MongoDB Atlas was restricted to the free-tier plan, which imposes a 512 MB storage limit. This significantly constrains the volume of data that could be tested for distributed storage and prevents comprehensive validation of large-scale distributed operations.
- **Lack of Real-Time Integration:** The current architecture does not yet incorporate real-time interfaces or APIs to integrate with external systems such as payment networks, regulatory services, or third-party fintech platforms, which are crucial in a production-ready digital wallet ecosystem.
- **Security and Compliance Limitations:** The implementation focuses on data consistency and integrity at the database level. However, advanced security measures—such as data encryption, secure user authentication, role-based access controls, auditing mechanisms, and regulatory compliance protocols (e.g., PCI DSS, GDPR)—are not implemented in this prototype. These aspects are critical for a production-grade fintech application.
- **Scalability Testing Constraints:** While concurrency and insertion performance were tested with thousands of records, large-scale stress testing involving millions of concurrent transactions could not be performed due to resource and environment limitations. Therefore, the system's true capacity for handling peak loads remains unverified.
- **Absence of User Interface and Administrative Tools:** The project focused exclusively on backend architecture and database design. Reporting dashboards, administrative panels, and user-facing applications were not developed. As a result, functionalities such as transaction visualizations, user management interfaces, and real-time analytics are not yet available.

- **Simplified Business Logic:** Some business rules were simplified for the scope of the academic exercise. Complex regulatory validations, fraud detection mechanisms, risk analysis, and credit scoring processes were not implemented, although they are essential in real fintech platforms.
- **Limited Historical Data Handling:** While the system successfully demonstrated the migration of older records to MongoDB Atlas, the limited data volume tested prevents validation of performance, search capabilities, and retrieval times in a true big data scenario involving several years of historical records.
- **Single Environment Deployment:** The system was developed and tested in a local environment or cloud testing environment. No deployment was carried out in a production-like distributed infrastructure with multiple regions, load balancing, or failover scenarios.

These limitations do not diminish the value of the progress achieved but serve as important considerations for future development phases. Addressing these gaps would be essential for transitioning Kine from a prototype to a production-ready fintech platform capable of operating at the scale and security levels required in the financial industry.

Chapter 8

Methodology

The methodology implemented for the development of the Kine project involved a series of structured steps aimed at designing and testing a hybrid database system suitable for a fintech environment. Initially, the project focused on understanding the business context through tools like the Business Model Canvas and gathering functional requirements. Subsequent phases included conceptual modeling, selection of database technologies, and the definition of an architecture capable of integrating relational and NoSQL systems. The implementation stage encompassed the development of Python scripts for data generation, database integration, and the creation of automated processes to handle large data volumes efficiently. Throughout these phases, emphasis was placed on designing a system capable of managing high transaction loads while maintaining scalability and data consistency.

8.1 Business Model Canvas

The project began with the development of a *Business Model Canvas* to understand the key components of Nequi's business model. This included identifying key partners, strategic resources, communication channels, value propositions, customer segments, and revenue streams. This analysis provided the foundation for identifying the core processes that the system architecture must support.

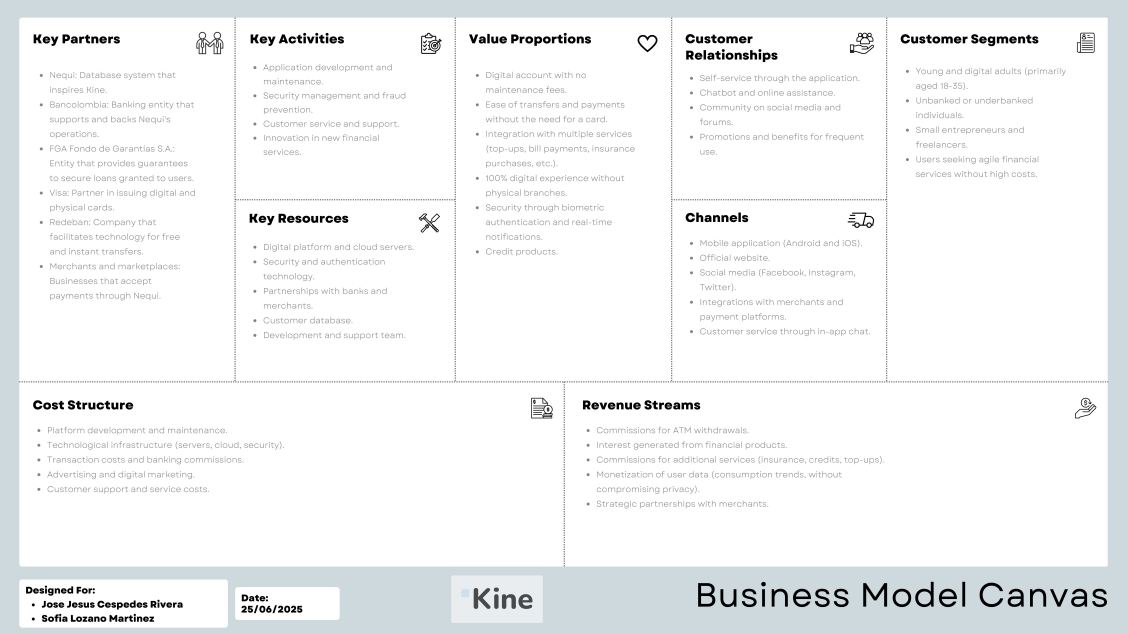

Figure 8.1: Business Model Canvas for Nequi

8.2 Requirements Elicitation

Based on the business model analysis, both functional and non-functional system requirements were identified and documented. Functional requirements described essential capabilities such as user registration, account management, money transfers, fund validation, transaction traceability, and customer support. Non-functional requirements addressed critical aspects such as system performance, scalability, availability, security, data integrity, and maintainability.

8.2.1 Functional Requirements

User Registration

The system must store the user's basic information (name, ID number, email, mobile number, etc.).

An account associated with the user must be registered with a unique identifier and status (active/inactive).

Account and Balance Management

The system must register multiple accounts associated with the same user (savings accounts, pockets, etc.).

The available balance of each account must be updated after every transaction.

Transaction Management

The system must register transactions of income type (top-ups, received transfers) and expense type (withdrawals, payments, sent transfers).

The transaction history must be stored with the following fields: ID, date and time, type, amount, origin, destination, status.

Funds Validation

Before registering an expense transaction, the system must verify that the account has sufficient funds.

Transfer Control

The system must store transfers between users (internal) and to other banks (external), with a status (pending, successful, failed).

Data Audit

An audit log must be maintained for sensitive changes: modifications to personal data, account deletions, transaction reversals.

Support for Multiple Devices/Sessions

The system must record from which device and approximate location the transactions originate for further analysis.

8.2.2 Non-Functional Requirements

Performance

Read and write operations (check balance, register transaction) must be performed in less than 1 second in 95% of the cases.

Scalability

The system must be able to handle millions of users and concurrent transactions, thus it must support horizontal and vertical partitioning of databases.

Availability

The system must be available 24/7 and must be designed with data backup (replication) and failure recovery.

Security

All sensitive information must be stored encrypted (e.g., hashed passwords, access tokens).

Transactions must include associated authentication (e.g., temporary token or session ID).

There must be role-based access control (user, system administrator, auditor, etc.).

Data Integrity

Transactions must comply with atomicity, consistency, isolation, and durability (ACID).

No transaction should be allowed to remain in an intermediate state.

Audit and Traceability

It must be possible to reconstruct the complete history of each user's operations for legal or security purposes.

Maintainability

The database structure must be normalized and documented to facilitate understanding and maintenance.

8.3 User Story Design

User stories were created following agile methodology to represent key system functionalities from the perspective of the end user. Each story included its priority, effort estimation, and acceptance criteria, allowing for effective organization and prioritization of functionalities based on user value.

In this project, user story estimation is carried out using the Story Points technique, on a discrete scale from 1 to 5:

- Very simple story, low effort, no dependencies or risks.
- Simple story, minimal logic or validations required.
- Medium-complexity story, includes business logic, validations, and testing.
- Complex story, involves multiple steps or interaction with external services.
- Very complex story, high technical risk or strong dependencies on other systems.

Priority is assigned based on the value delivered to the end user, the urgency of the requirement, and its impact on the core user flow of the application. For this project, we use a three-level scale:

- High: Critical or core functionality for the basic use of Nequi (e.g., registration, balance top-up, money transfer).
- Medium: Important but non-blocking features (e.g., bill payments, savings goals).
- Low: Complementary or less urgent features (e.g., customization, non-essential transaction history).

Table 8.1: User Story US-01

Title	**Priority**	**Estimate (1-5)**
Account registration and creation	High	2 points
User Story: As a user, I want to register and create a Nequi account so that I can manage my finances without complications.		
Acceptance Criteria: Given the user downloads the Nequi app, when they complete the registration form and accept the terms, then the system creates an account and notifies successful registration.		

Table 8.2: User Story US-02

Title	**Priority**	**Estimate (1-5)**
Balance Top-Up	High	2 points
User Story: As a user, I want to top up my balance so that I have funds available for transactions.		
Acceptance Criteria: Given the user has an active account, when they select the top-up option and provide amount and method, then the system processes and confirms the transaction.		

Table 8.3: User Story US-03

Title	**Priority**	**Estimate (1-5)**
Send & Receive Money	High	3 points
User Story: As a user, I want to send and receive money so that I can transfer funds quickly and securely.		
Acceptance Criteria: Given the user has balance, when they enter recipient details and confirm, then the system processes and notifies both parties.		

Table 8.4: User Story US-04

Title	**Priority**	**Estimate (1-5)**
Payments in Stores	Medium	3 points
User Story: As a user, I want to pay at stores so that I can shop without cash or physical cards.		
Acceptance Criteria: Given the store accepts Nequi, when I scan or enter the merchant code, then the system deducts the amount and notifies both parties.		

Table 8.5: User Story US-05

Title	**Priority**	**Estimate (1-5)**
Expense & Savings Management	Medium	3 points
User Story: As a user, I want to manage expenses and savings so that I can control my finances and reach goals.		
Acceptance Criteria: Given the user wants a savings goal, when they provide name, amount and deadline, then the system creates the goal and tracks progress.		

Table 8.6: User Story US-06

Title	**Priority**	**Estimate (1-5)**
Credit Application	Medium	4 points
User Story: As a user, I want to apply for a loan so that I can get financing under favorable conditions.		
Acceptance Criteria: Given the user meets the requirements, when they apply for a loan and enter the amount and term, then the system evaluates and disburses funds if approved.		

Table 8.7: User Story US-07

Title	**Priority**	**Estimate (1-5)**
Mobile Top-Up & Bill Payments	High	3 points
User Story: As a user, I want to pay services and top up from one platform so that I can manage everything easily.		
Acceptance Criteria: Given the user wants to pay a bill, when they select the service, enter details, and confirm, then the system processes and notifies successful payment.		

Table 8.8: User Story US-08

Title	**Priority**	**Estimate (1-5)**
Security and Support	High	5 points
User Story: As a user, I want my transactions to be protected and have support when needed so that I feel secure.		
Acceptance Criteria: Given the user logs in from a new device, when the system asks for biometric or code verification, then the user authenticates and securely accesses their account.		

8.4 Data Architecture Design

The data architecture for Kine was designed around five core components: user and account management, financial transaction management, loan management, authentication and security, and user support. Based on these components, the main entities were defined, along with their attributes, primary and foreign keys, and inter-entity relationships. This design process followed a methodology inspired by ontology development principles derived from the Entity-Relationship (ER) modeling approach, which ensured a systematic identification of system components, entities, attributes, relationships, cardinalities, and constraints.

Throughout the design, many-to-many relationships were analyzed and transformed into associative entities where necessary to maintain data normalization. The outcome of this process is a finalized ER model that accurately represents the data structure required for the system's implementation, providing clear definitions of entities, relationships, and business rules to ensure data integrity and consistency across the application.

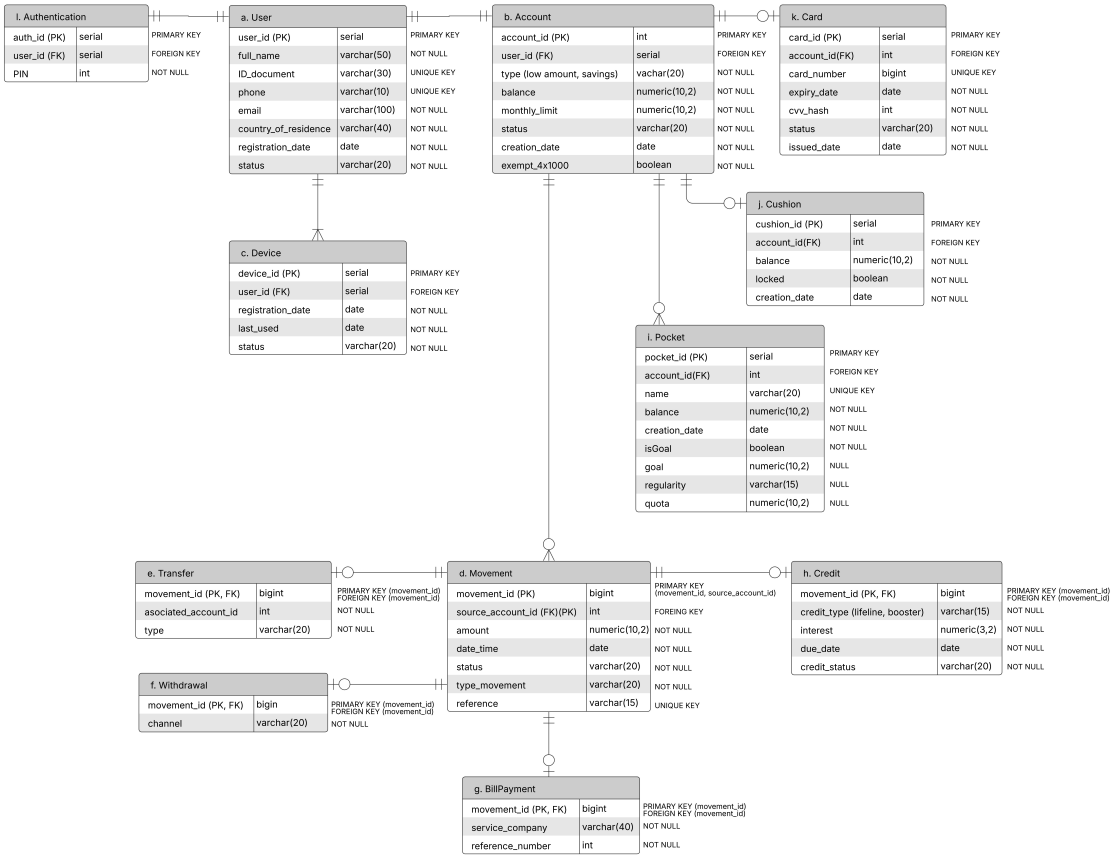

Figure 8.2: Entity-Relationship Model

8.5 Data System Structure

The structure of the data system was designed to support the operational, transactional, and support functionalities required by Kine. A relational model was initially established using PostgreSQL, where core entities such as users, accounts, transactions, and authentication records were normalized to ensure consistency, reduce redundancy, and support ACID-compliant operations.

To handle large volumes of semi-structured and historical data, such as transaction logs and user support requests, the system was extended to include MongoDB Atlas as a distributed NoSQL database. This hybrid architecture allows for the separation of frequently updated transactional data and infrequently accessed archival data, improving performance and scalability.

The decision to integrate both relational and non-relational databases was driven by the need to optimize query efficiency, support data concurrency, and enable future extensions such as real-time analytics and extract generation. This structure ensures that each type of data is stored and processed in the most appropriate environment according to its nature and usage frequency.

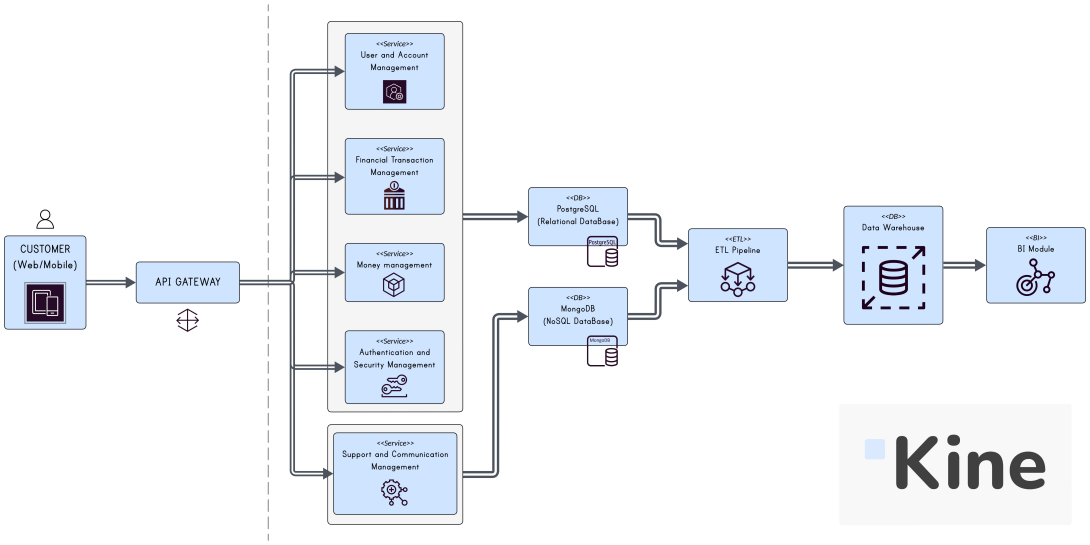

Figure 8.3: Data system structure

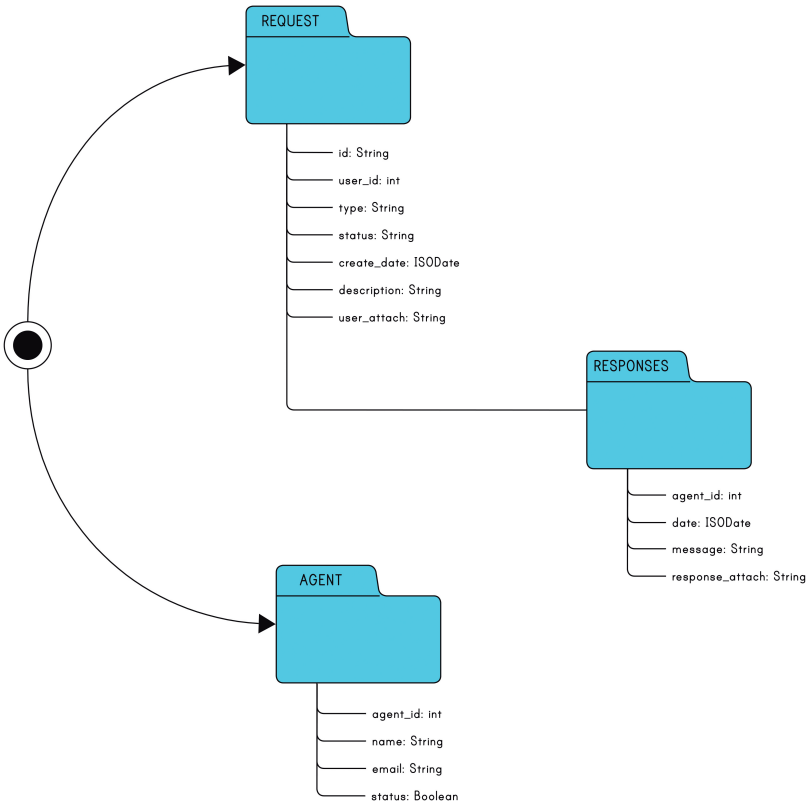

Figure 8.4: Model NoSQL

8.6 Information Sources Kine

The data architecture of Kine is supported by various information sources that reflect the different channels and operations of the digital wallet ecosystem. The primary sources identified include the mobile application, ATMs, banking correspondents, and companies interacting with the platform. Each of these sources contributes distinct types of data:

- **App:** Captures data related to users, services, financial movements, and support interactions. This includes information such as user profiles, account details, devices, authentication records, service types (e.g., cards, cushions, pokens), transaction details like transfers, bill payments, credits, and support requests with descriptive data.
- **ATM:** Provides information about withdrawal operations, including basic details of the withdrawal itself, the user performing the transaction, and the ATM used.
- **Banking Correspondent:** Records data related to withdrawals executed through third-party banking services, associating transactions with both the user and the correspondent entity.
- **Companies:** Supplies information about bill payment transactions, detailing the relationship between users and the specific companies involved.

The integration of these sources ensures that the system captures comprehensive data covering all operational and transactional scenarios, enabling accurate tracking, reporting, and support for the services provided by Kine.

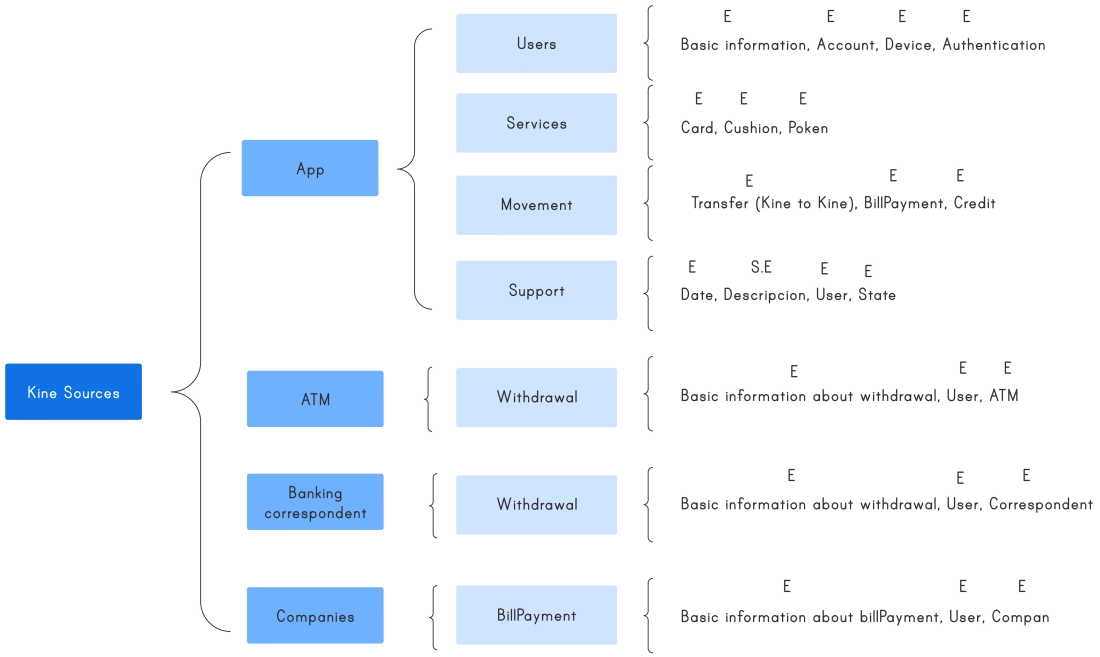

Figure 8.5: Data system structure

8.7 Concurrency Analysis

Concurrency scenarios were analyzed to ensure data consistency and system reliability under simultaneous operations typical in financial environments. Several potential issues were iden-

tified and addressed through targeted solutions integrated into the system's design.
Simultaneous transfers and payments from the same account could result in negative balances if both operations read the same initial data. To mitigate this, pessimistic locking was implemented on account records, allowing only one operation to modify the balance at a time.
For cases where multiple users transfer funds concurrently to the same account, PostgreSQL's native Multiversion Concurrency Control (MVCC) prevents dirty reads, allowing reads and writes to occur without blocking each other.
Background tasks like automated payments or scheduled debits might access the same records simultaneously with user actions. Appropriate transaction isolation levels were applied, with Read Committed or Repeatable Read used in most cases, and Serializable reserved for critical operations requiring higher consistency.
Concurrent use of savings Pockets, such as automatic deposits overlapping with manual withdrawals, was addressed using optimistic concurrency control through versioning. This detects conflicting changes and forces a retry to maintain data integrity.
Finally, to prevent duplicate transactions caused by user retries—like pressing "Send" multiple times after a network delay—the system employs idempotency using unique UUIDs for each transaction. The backend checks whether a transaction with the same UUID already exists and avoids executing it twice.
These mechanisms enhance Kine's robustness and reliability, ensuring data integrity even under high concurrency conditions.

8.8 Parallel and Distributed Database Design

The design of Kine's data architecture was guided by the operational requirements of a financial platform handling high transaction volumes and stringent regulatory compliance. Although parallel database techniques are suitable for OLAP systems focused on analytical queries and batch processing, they were not deemed necessary for Kine, whose workload is primarily transactional (OLTP) with real-time demands. Instead, the project adopted a distributed architecture to enhance availability, scalability, and regulatory compliance.

8.8.1 Distributed Architecture Rationale

- **Transactional Focus:** Kine's core operations include frequent, small transactions such as transfers, payments, and balance inquiries, prioritizing data integrity, consistency, and immediate availability.
- **Regulatory Requirements:** Colombian financial regulations demand that transaction histories remain accessible for at least two months, after which older data can be archived. A distributed system enables seamless separation between operational (hot) data and historical (cold) storage.
- **Scalability and Fault Tolerance:** Distributed databases allow horizontal scaling and replication across regions, providing resilience and maintaining ACID guarantees for critical financial operations.

8.8.2 High-Level Architecture Design

- **API Gateway:** All user interactions pass through the API Gateway, which manages authentication, routes requests to either the transactional system or the historical archive, and centralizes security controls.

- **Transactional Database Cluster:** Active transactions and current balances are stored in a high-performance relational database cluster (PostgreSQL), featuring synchronous replication to ensure high availability and fault tolerance.
- **ETL Processes:** Periodic ETL operations extract data older than two months, transform and compress it as necessary, and migrate it to cold storage.
- **Cold Storage:** Older transaction records are archived in a cloud-based system, reducing costs and minimizing the load on the transactional database while remaining accessible for audits and regulatory queries.
- **Historical Queries:** Requests for data beyond the active period are routed by the API Gateway to the cold storage system. Business Intelligence tools and auditors can query this archive directly without impacting live operations.
- **Monitoring and Coordination:** Continuous monitoring ensures data integrity, detects errors, and verifies that replication and archiving processes run smoothly.

8.8.3 Reasons Against Parallel Database Use

- Kine does not require large-scale analytical queries typical of OLAP systems.
- Introducing parallel database processing would add unnecessary complexity and cost without substantial benefits for Kine's OLTP workload.
- The distributed design sufficiently covers the needs for high availability, scalability, and compliance, rendering parallelism redundant in this context.

This approach allows Kine to maintain real-time responsiveness for operational transactions while complying with legal requirements for data retention and optimizing infrastructure costs.

8.9 Performance Improvement Strategies

To ensure that Kine's architecture can support high transaction volumes and maintain optimal performance, several performance improvement strategies were considered and incorporated into the system design.

8.9.1 Indexing and Query Optimization

Indexes were implemented on frequently queried columns, such as user identifiers, account numbers, and transaction dates, to accelerate searches and reduce query latency. Careful analysis of query patterns guided the decision to create composite indexes where necessary, avoiding unnecessary indexes that could increase write overhead.

8.9.2 Use of Batch Operations

Batch operations were applied for data insertions and updates, especially during the migration of historical data to MongoDB Atlas. This approach minimizes the number of database round-trips and improves throughput when processing large datasets.

8.9.3 Transaction Isolation Levels

Appropriate transaction isolation levels were selected to balance consistency and performance. For standard operations, Read Committed or Repeatable Read was used to reduce locking contention, while critical financial operations leveraged Serializable transactions to ensure maximum consistency.

8.9.4 Connection Pooling

Connection pooling was configured for both PostgreSQL and MongoDB to reduce connection overhead and improve response times under concurrent workloads.

8.9.5 ETL Scheduling and Load Balancing

The ETL processes responsible for migrating data older than two months to cold storage were scheduled during off-peak hours to avoid impacting the performance of active transactional workloads. Additionally, load balancing strategies were designed to distribute query traffic across replicas where possible.

8.9.6 Data Partitioning

Considerations were made for future data partitioning strategies in PostgreSQL, particularly for large tables like transaction histories. Partitioning by date or account ID can significantly improve query performance and reduce table scan times.

8.9.7 Idempotency and Request Management

To prevent duplicated operations due to network retries or user actions, idempotency mechanisms using UUIDs were implemented for financial transactions. This reduces unnecessary workload and maintains data integrity.

These combined strategies enhance Kine's ability to handle large-scale operations efficiently while ensuring data consistency, scalability, and compliance with financial regulations.

Chapter 9

Results

This chapter presents the detailed results of the Kine system implementation and testing. The findings cover the system's architecture, the processes used for generating and inserting synthetic data, the performance of transactional operations, the application of database triggers, the extraction of user financial data, and the integration with distributed NoSQL storage. Overall, the project aimed to validate the feasibility and performance of a hybrid architecture capable of handling the demands of a modern fintech environment.

9.1 Generation of Users and Accounts

To simulate realistic operational conditions, the project required a substantial dataset of users and accounts. Python was used to develop scripts that automatically generate this data. The Faker library was instrumental in creating realistic user attributes such as names, addresses, email addresses, phone numbers, and dates of birth. Additionally, the `random` module was employed to introduce variability in certain fields, ensuring that the data reflected diverse user behaviors and account configurations.
The generated data included essential details for each user, as well as their associated accounts. This allowed for testing of entity relationships and referential integrity within the PostgreSQL database. Moreover, `pandas` was used to manipulate and structure large amounts of data before insertion into the database, enabling batch operations that significantly reduced the time needed for data loading.
A performance test was conducted involving the insertion of 1,000 user records and their corresponding account records. The results confirmed that the database could handle the creation of a large volume of data in a short period of time, demonstrating good performance and validating the entity-relationship model implemented during the design phase.

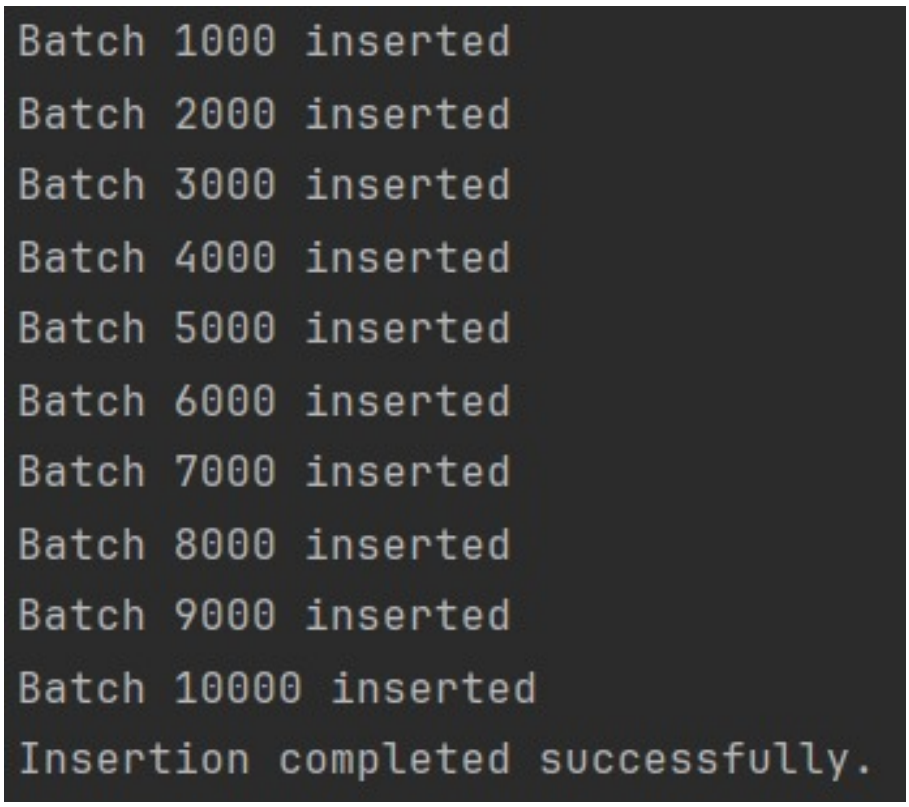

Figure 9.1: Insertion test for 1,000 users and accounts

9.2 Insertion of Movements and Performance

Testing continued with the generation of synthetic financial movements linked to the previously created user accounts. Using Faker and random number generation, transaction amounts, transaction types (transfer, withdrawal, bill payment, credit), timestamps, and other related fields were created to mimic real financial behavior. This allowed the system to be tested under realistic load conditions.

A performance test was executed in which 463 financial transaction records were inserted into the database. The process completed in approximately 273 milliseconds, significantly below the target threshold of one second for critical OLTP operations. This result underscores the efficiency of PostgreSQL in handling bulk inserts and transactional workloads, which is crucial for fintech platforms that process high volumes of financial operations.

Furthermore, this test verified that the constraints, triggers, and relationships defined in the schema correctly enforced data consistency. No errors were encountered during the insertion process, indicating that the data model was robust under operational load.

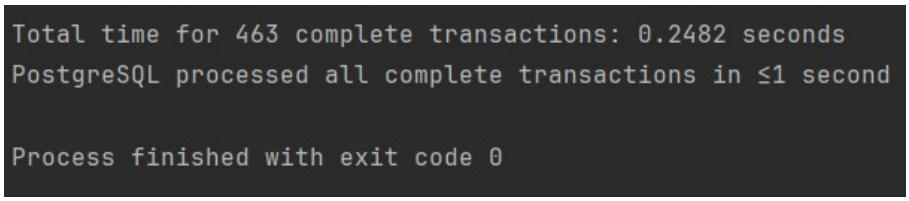

Figure 9.2: Insertion test for transaction records and execution time

9.3 Entity-Relationship Model Implementation

A central outcome of the project was the transformation of the conceptual entity-relationship model into physical SQL code for PostgreSQL. The ER model defines the structure of tables, primary and foreign key constraints, and relationships between entities such as users, accounts, transactions, support requests, and more.

This step translates the logical design into a functional database schema that ensures data integrity, consistency, and the enforcement of business rules. Figure 9.3 illustrates the SQL scripts developed to create the relational schema in PostgreSQL.

The SQL code reflects the results of the methodological steps described earlier in the report, including the ontology-based modeling and normalization of relationships. This stage was essential for ensuring that all subsequent operations—including inserts, updates, queries, and reporting—could execute reliably and efficiently.

```sql
CREATE EXTENSION IF NOT EXISTS pgcrypto;
-- ============================================
-- Tabla: User
-- ============================================
CREATE TABLE "user" (
    user_id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    full_name VARCHAR(100) NOT NULL,
    ID_document VARCHAR(15) NOT NULL UNIQUE,
    phone VARCHAR(10) NOT NULL UNIQUE,
    email TEXT NOT NULL UNIQUE,
    country_of_residence VARCHAR(50),
    registration_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    status VARCHAR(15) NOT NULL CHECK (status IN ('active', 'inactive'))
);

-- ============================================
-- Tabla: Authentication
-- ============================================
CREATE TABLE "authentication" (
    auth_id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID UNIQUE NOT NULL,
    PIN INT NOT NULL CHECK (PIN BETWEEN 0 AND 9999),
    FOREIGN KEY (user_id) REFERENCES "user"(user_id) ON DELETE CASCADE
);

-- ============================================
-- Tabla: Device
-- ============================================
CREATE TABLE "device" (
    device_id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID NOT NULL,
    registration_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    last_used TIMESTAMP,
    status VARCHAR(15) NOT NULL CHECK (status IN ('active', 'inactive', 'blocked')),
    FOREIGN KEY (user_id) REFERENCES "user"(user_id) ON DELETE CASCADE
);

-- ============================================
-- Tabla: Account
-- ============================================
CREATE TABLE "account" (
    account_id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID NOT NULL UNIQUE,
    type_account VARCHAR(20) NOT NULL CHECK (type_account IN ('low_amount', 'savings')),
    balance NUMERIC(15, 2) DEFAULT 0 CHECK (balance >= 0),
    monthly_limit NUMERIC(15,2) CHECK (monthly_limit >= 0),
    status VARCHAR(15) NOT NULL CHECK (status IN ('active', 'inactive', 'blocked')),
    creation_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    exempt_4x1000 BOOLEAN DEFAULT FALSE,
    FOREIGN KEY (user_id) REFERENCES "user"(user_id) ON DELETE CASCADE
);

-- ============================================
-- Tabla: Movement
-- ============================================
CREATE TABLE "movement" (
    movement_id serial PRIMARY KEY,
    source_account_id UUID NOT NULL,
    amount NUMERIC(15,2) NOT NULL CHECK (amount >= 0),
    date_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    status VARCHAR(15) NOT NULL CHECK (status IN ('pending', 'completed', 'failed')),
    type_movement VARCHAR(20) NOT NULL CHECK (type_movement IN ('transfer', 'withdrawal', 'bill_payment', 'credit')),
    reference VARCHAR(50),
    FOREIGN KEY (source_account_id) REFERENCES "account"(account_id) ON DELETE CASCADE
);

-- ============================================
-- Tabla: Transfer
-- ============================================
CREATE TABLE "transfer" (
    movement_id integer PRIMARY KEY,
    associated_account_id UUID NOT NULL,
    type_transfer VARCHAR(15) NOT NULL CHECK (type_transfer IN ('internal', 'external')),
    FOREIGN KEY (movement_id) REFERENCES "movement"(movement_id) ON DELETE CASCADE,
    FOREIGN KEY (associated_account_id) REFERENCES "account"(account_id) ON DELETE CASCADE
);
```

Figure 9.3: SQL code for table creation based on the ER model (1)

```sql
-- ============================================
-- Tabla: Withdrawal
-- ============================================
CREATE TABLE "withdrawal" (
    movement_id integer PRIMARY KEY,
    channel VARCHAR(30) NOT NULL CHECK (channel IN ('ATM', 'branch', 'corresponsal', 'online')),
    FOREIGN KEY (movement_id) REFERENCES "movement"(movement_id) ON DELETE CASCADE
);

-- ============================================
-- Tabla: BillPayment
-- ============================================
CREATE TABLE "billPayment" (
    movement_id integer PRIMARY KEY,
    service_company VARCHAR(100) NOT NULL,
    reference_number VARCHAR(50) NOT NULL,
    FOREIGN KEY (movement_id) REFERENCES "movement"(movement_id) ON DELETE CASCADE
);

-- ============================================
-- Tabla: Credit
-- ============================================
CREATE TABLE "credit" (
    movement_id integer PRIMARY KEY,
    credit_type VARCHAR(20) NOT NULL CHECK (credit_type IN ('lifeline', 'booster')),
    interest NUMERIC(5,2) NOT NULL CHECK (interest >= 0),
    due_date DATE NOT NULL,
    credit_status VARCHAR(20) NOT NULL CHECK (credit_status IN ('approved', 'pending', 'rejected')),
    FOREIGN KEY (movement_id) REFERENCES "movement"(movement_id) ON DELETE CASCADE
);

-- ============================================
-- Tabla: Pocket
-- ============================================
CREATE TABLE "pocket" (
    pocket_id serial PRIMARY KEY,
    account_id UUID NOT NULL,
    name_pocket VARCHAR(50) NOT NULL,
    balance NUMERIC(15,2) DEFAULT 0 CHECK (balance >= 0),
    creation_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    is_goal BOOLEAN NOT NULL DEFAULT FALSE,
    goal NUMERIC(15,2) CHECK (goal >= 0),
    regularity VARCHAR(20) CHECK (regularity IN ('daily', 'weekly', 'biweekly', 'monthly')),
    quota NUMERIC(15,2) CHECK (quota >= 0),
    FOREIGN KEY (account_id) REFERENCES "account"(account_id) ON DELETE CASCADE
);

-- ============================================
-- Tabla: Cushion
-- ============================================
CREATE TABLE "cushion" (
    cushion_id serial PRIMARY KEY,
    account_id UUID NOT NULL UNIQUE,
    balance NUMERIC(15,2) DEFAULT 0 CHECK (balance >= 0),
    is_locked BOOLEAN DEFAULT FALSE,
    creation_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (account_id) REFERENCES "account"(account_id) ON DELETE CASCADE
);

-- ============================================
-- Tabla: Card
-- ============================================
CREATE TABLE "card" (
    card_id serial PRIMARY KEY,
    account_id UUID NOT NULL UNIQUE,
    card_number VARCHAR(20) NOT NULL UNIQUE,
    expiry_date DATE NOT NULL,
    status VARCHAR(15) NOT NULL CHECK (status IN ('active', 'inactive', 'cancelled')),
    FOREIGN KEY (account_id) REFERENCES "account"(account_id) ON DELETE CASCADE
);
```

Figure 9.4: SQL code for table creation based on the ER model (2)

9.4 Implementation of Database Triggers

During the implementation phase, several triggers were identified as necessary to enforce business logic directly at the database level. A key example is the trigger created for the `pocket` table, which ensures that when a record is marked as a savings goal (`is_goal = TRUE`), critical fields such as `goal`, `regularity`, and `quota` must not be null. This prevents incomplete or inconsistent data from being inserted or updated in the database.

The implementation of triggers helps maintain data integrity and reduces reliance on application-layer validations alone. It ensures that the business rules are enforced consistently, regardless of the source of the database operation.

Figure 9.5 shows a screenshot of the trigger code within the development environment, high-

lighting the practical implementation of database-level validations.

```sql
-- ========================================
-- Trigger Function for pocket
-- ========================================
CREATE OR REPLACE FUNCTION validate_pocket_goal_fields()
RETURNS TRIGGER AS $$
BEGIN
  -- Si es una meta de ahorro
  IF NEW.is_goal = TRUE THEN
    IF NEW.goal IS NULL OR NEW.regularity IS NULL OR NEW.quota IS NULL THEN
      RAISE EXCEPTION 'Campos goal, regularity y quota no pueden ser NULL cuando is_goal = TRUE';
    END IF;
  END IF;

  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- ========================================
-- Trigger for pocket
-- ========================================
CREATE TRIGGER trg_validate_pocket_goal
BEFORE INSERT OR UPDATE ON "pocket"
FOR EACH ROW
EXECUTE FUNCTION validate_pocket_goal_fields();
```

Figure 9.5: Code for trigger function to validate pocket goal fields

9.5 Generation of User Extracts

A critical functionality in financial systems is the ability to generate extracts or statements for individual users over specific time periods. To implement this, Python scripts were developed using pandas for data manipulation and formatting. The scripts query the database to retrieve all transactions associated with a given user within a defined date range.
The extract includes a detailed list of transactions, including dates, transaction types, statuses, references, and amounts. Additionally, the scripts calculate the total sum of incoming funds (credits) and outgoing funds (debits) for the user during the selected period. This provides valuable insights into user behavior and financial status.
Figure 9.6 illustrates an example of a generated user extract, demonstrating the system's ability to produce detailed and accurate reports suitable for both end users and regulatory compliance.

statement_Ángela_Viviana_Restrepo_Forero_2025-02-01_to_2025-02-28.csv

Origen de archivo: 1252: Europeo occidental (Windows)
Delimitador: Coma
Detección del tipo de datos: Basado en las primeras 200 filas

date_time	type_movement	status	reference	amount
2025-02-25 04:58:56	credit	completed	Trans875Mhx	*298633291*
				null
Total Income	2986332.91			*null*
Total Expense	0			*null*

Figure 9.6: Generated user extract showing detailed transactions and totals

9.6 Integration with MongoDB Atlas

As part of the distributed architecture, MongoDB Atlas was integrated to store semi-structured data, particularly for support operations such as user requests and agent responses. Although the project was constrained to the 512 MB free-tier limit, the experiments successfully validated the integration between PostgreSQL and MongoDB Atlas.
Data for the `request` and `response` collections was generated automatically using Python scripts. These scripts used Faker to create realistic support scenarios, generating random users, support topics, agent responses, timestamps, and other related fields.
The integration confirmed that MongoDB Atlas can serve as a distributed storage solution, capable of handling documents related to support interactions or historical transaction records. Figures 9.7 and 9.8 show the data successfully inserted and stored in MongoDB Atlas, demonstrating both the technical feasibility and practical implementation of a hybrid architecture.

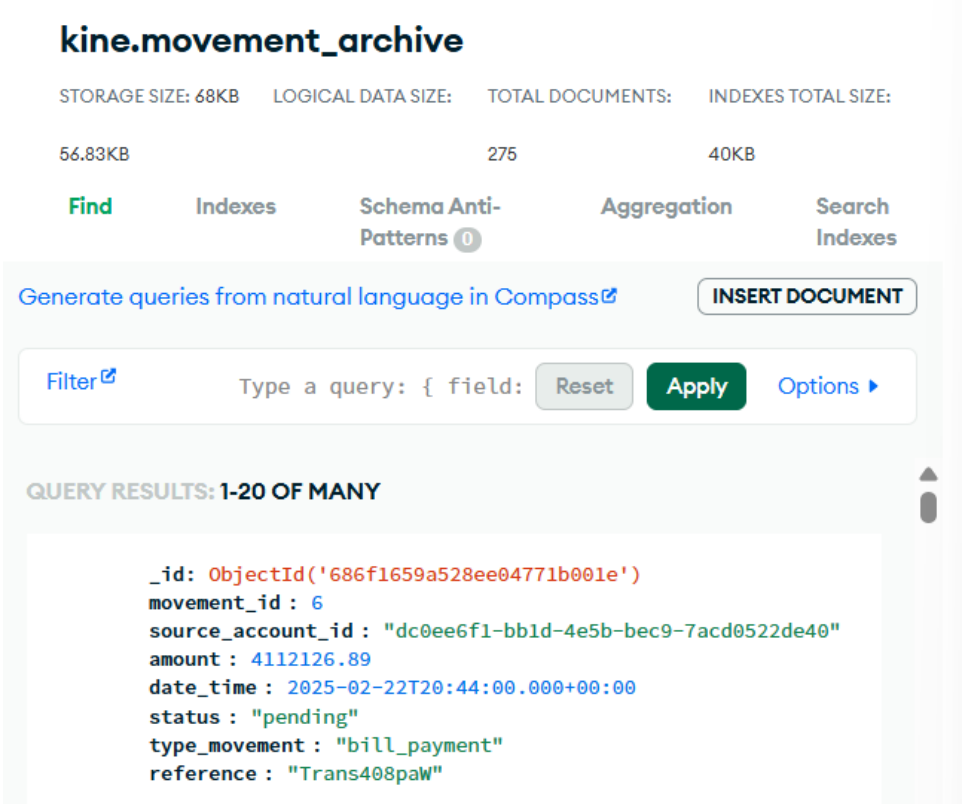

Figure 9.7: MongoDB Atlas showing transferred data from Kine

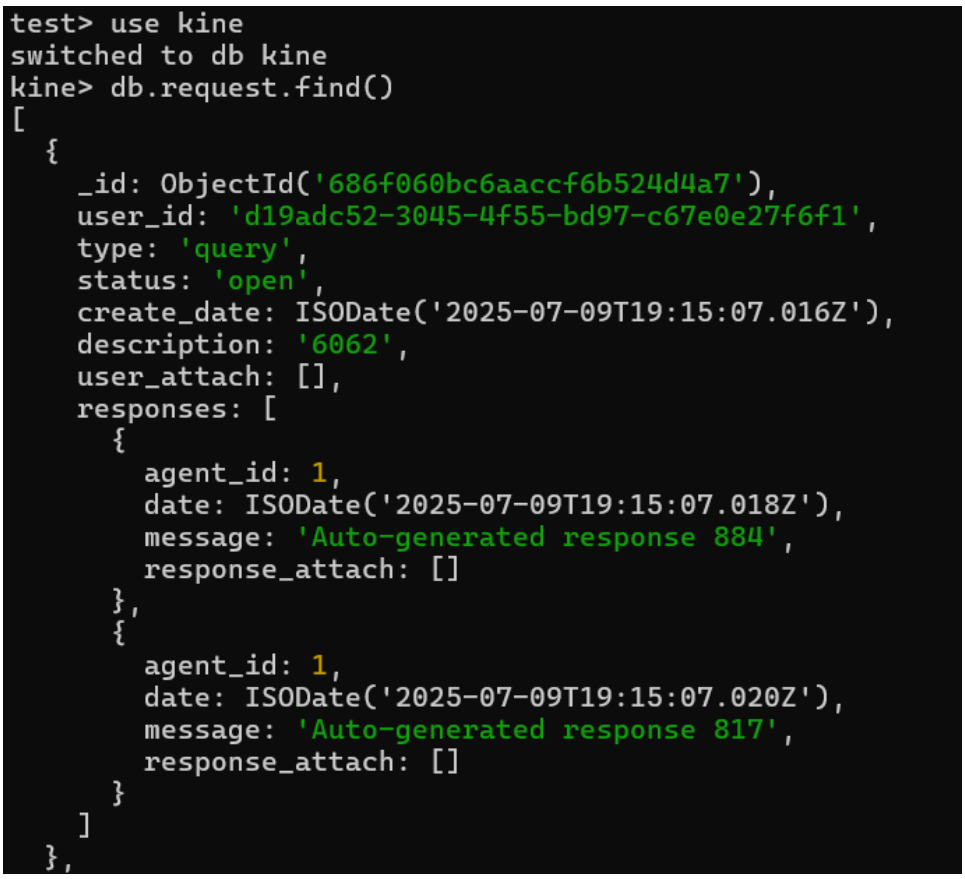

Figure 9.8: Insertion of automatically generated request and response documents in MongoDB Atlas

9.7 Summary of Results

The results of the Kine project validate the feasibility of implementing a hybrid architecture using PostgreSQL for transactional operations and MongoDB Atlas for semi-structured and distributed data storage. The system performed well in high-volume insertion scenarios and demonstrated strong data integrity through the use of triggers and constraints.

The integration with MongoDB Atlas confirmed the possibility of expanding the system to manage larger volumes of historical or support-related data in a distributed manner, even though limited testing was conducted due to free-tier storage restrictions.

Overall, the developed architecture and the performed tests indicate that Kine's design can support the demanding requirements of a digital banking environment, providing both robust transactional performance and flexibility for future scalability.

Chapter 10

Discussion

The development process of the Kine project provided valuable insights into the technical and architectural considerations necessary to build a functional digital banking platform. Throughout the implementation, several key decisions, challenges, and reflections emerged, which are discussed in this section.

10.1 Ontology-Based Modeling and Design Consistency

The use of the "10 Steps of Ontology" methodology proved to be instrumental in structuring the knowledge domain of digital financial services. By following a methodical process—from defining system components to drawing entity-relationship diagrams and setting data constraints—the project maintained consistency, normalization, and semantic clarity across the database structure.

Mapping user stories to data components ensured alignment between the conceptual design and functional requirements. Each operation envisioned in the business logic (e.g., transfers, payments, pocket management, and support requests) was explicitly mapped to tables, constraints, and triggers, thereby avoiding ambiguous or redundant structures.

10.2 Modular Transaction Design and Hierarchical Abstractions

One of the most impactful architectural outcomes was the modular and hierarchical design of financial transactions. Initially, overlapping operations such as transfers, bill payments, and credits introduced potential redundancy and confusion. To resolve this, the system implemented a superclass-subclass structure that grouped these operations under a unified schema, with specific attributes handled through enumerated types and optional fields. This approach supports scalability while preserving data integrity.

In doing so, the model anticipates future needs, allowing for specialized transaction types without modifying core table definitions. This design contrasts with rigid legacy banking systems, where every operation often requires a separate table or hardcoded logic.

10.3 Use of PostgreSQL and NoSQL in Hybrid Architecture

The decision to adopt PostgreSQL for core transactional data was supported by its robustness, ACID compliance, and concurrency control mechanisms such as MVCC and transaction isolation levels. PostgreSQL's native capabilities allowed the team to enforce business logic through

constraints and triggers (e.g., pocket savings goals), reducing dependency on application-layer validations.

At the same time, MongoDB Atlas was introduced to support the storage of semi-structured data—particularly support-related documents (requests and responses)—and to validate the integration of distributed, horizontally scalable storage mechanisms. Although limited by the free-tier capacity (512 MB), this integration confirmed that the architecture can evolve toward a distributed and modular environment.

The hybrid approach mirrors current trends in modern data architecture, where relational systems are complemented by NoSQL databases to optimize storage, retrieval, and scalability based on data type and use case.

10.4 Data Generation and Testing Realism

The use of Faker, pandas, and random in Python allowed the team to rapidly generate thousands of realistic records for users, accounts, transactions, and support tickets. This facilitated the simulation of operational scenarios that closely resemble the complexity of real fintech environments.

However, it was also acknowledged that synthetic data, while useful for functional validation and load testing, does not fully capture the behavioral nuances or edge cases found in production systems. An important lesson was the need to combine synthetic and real datasets (in future stages) for more comprehensive system evaluations.

10.5 Concurrency, Triggers, and Data Integrity

Concurrency emerged as a central challenge during implementation. Financial systems must support multiple operations—transfers, savings deposits, withdrawals—happening simultaneously without risking data inconsistency. Kine leveraged PostgreSQL's transaction management capabilities, implementing a combination of pessimistic locking, MVCC, and transaction isolation levels to handle these cases.

Additionally, database triggers played a critical role in enforcing business logic, such as validating savings goal configurations. This ensured that even if application-level controls fail or are bypassed, the database remains consistent and trustworthy. Such design patterns reflect industry best practices in system resilience and data integrity.

10.6 Comparison with Legacy Banking Systems

Legacy banking systems often struggle with scalability, modularity, and integration. In contrast, the Kine model emphasizes separation of concerns, modular architecture, and distributed capabilities. The project demonstrates that even with academic constraints, it is possible to architect a modern system that balances consistency and flexibility.

Unlike monolithic database structures, Kine's model is ready to evolve into a microservices-oriented infrastructure, where each functional module (e.g., movement engine, support module, extract generation) could operate independently while sharing a common data backbone.

10.7 Identified Trade-offs and Future Considerations

Despite the project's achievements, several trade-offs were made. No user interface was developed; thus, the system operates at the backend and database levels only. Security

mechanisms were limited to structural validations, and high-scale distributed testing was not possible under the available environment.

However, these limitations served to highlight critical next steps:

- Integration with real-time APIs and third-party financial services.
- Full implementation of access control, encryption, and secure authentication protocols.
- Deployment in a cloud-based, replicated architecture using sharding or horizontal scaling (as described by MongoDB Inc. MongoDB Inc. (2024)).
- Inclusion of real datasets for behavior modeling and machine learning experiments.
- Development of dashboards and interfaces for both end users and administrative roles.

10.8 Conclusion of Discussion

The Kine project was not only a technical exercise but also an exploration of the decisions, compromises, and practices involved in designing a financial data architecture. It reaffirmed the importance of modeling clarity, modular structure, hybrid architectures, and system resilience. While academic in scope, Kine provides a realistic foundation from which a real-world, scalable fintech platform could emerge.

Chapter 11

Conclusion

The development of the Kine project has advanced beyond the initial design phase to include significant implementation and testing efforts. Throughout this work, a hybrid data architecture was conceived and validated, combining the strengths of PostgreSQL for transactional consistency with the flexibility of MongoDB Atlas for semi-structured and distributed data storage.

Applying the "10 Steps of Ontology" methodology was crucial in transforming business requirements into a well-structured conceptual and physical data model. The resulting entity-relationship model reflects the complexity of digital financial operations, ensuring normalization, integrity, and future scalability.

One of the most significant achievements of this phase was the successful generation and insertion of synthetic data using Python libraries such as Faker, pandas, and random. This allowed the simulation of thousands of users, accounts, and transactions, validating the system's capacity to handle realistic workloads. Performance testing demonstrated that the system can process hundreds of transactional inserts in under one second, aligning with the operational demands of high-traffic fintech platforms like Nequi, which processes more than 38 million transactions daily Henao (2025).

The integration of MongoDB Atlas, though limited to the free-tier capacity, confirmed the feasibility of distributing certain data types—such as support tickets and historical records—outside the primary transactional engine. This architecture paves the way for future scalability and big data analytics.

Equally important were the measures implemented to ensure data consistency and integrity. Concurrency challenges were addressed using PostgreSQL's transaction isolation, MVCC, and pessimistic locking strategies. Business rules were enforced through database triggers and constraints, ensuring that critical validations occurred at the data layer.

Nevertheless, several limitations remain. Security measures, such as encryption and advanced authentication, were not implemented in this prototype. The system was tested only in a controlled, non-production environment, and scalability to millions of concurrent users requires further validation. User-facing interfaces and administrative dashboards also remain to be developed.

Despite these limitations, the Kine project provides a solid foundation for building a scalable, modular, and secure digital wallet platform. The work completed so far demonstrates not only the feasibility of implementing a hybrid relational-NoSQL architecture but also the practical steps necessary to manage large-scale financial data operations. Future work will focus on extending this architecture, incorporating real-time integrations, developing user interfaces, enhancing security, and conducting large-scale performance tests to bring Kine closer to a production-ready fintech solution.

References

Henao, D. A. O. (2025), '"mediante nequi hacemos una marca de más de 38 millones de transacciones diarias"'. [Online].
URL: *https://www.larepublica.co/finanzas/entrevista-con-andres-vasquez-echeverri-ceo-de-nequi-sobre-las-transacciones-en-colombia-4039379/*

Jain, A. (2022), 'Scaling postgresql for high-traffic applications'. [Online].
URL: *https://www.percona.com/blog/scaling-postgresql-high-traffic-apps/*

MongoDB Inc. (2023), 'Mongodb vs. relational databases'. [Online].
URL: *https://www.mongodb.com/compare/mongodb-vs-relational-databases*

MongoDB Inc. (2024), 'Mongodb architecture guide: Sharding and scaling'. [Online].
URL: *https://www.mongodb.com/docs/manual/sharding/*

PostgreSQL Global Development Group (2023), 'Postgresql performance: Benchmarks and use cases'. [Online].
URL: *https://www.postgresql.org/docs/current/performance-tips.html*

PostgreSQL Global Development Group (2024), 'pgbench — run a benchmark test on postgresql'. [Online]. Recently updated. Typical performance of 896.97 TPS in default environments with 10 clients is mentioned.
URL: *https://www.postgresql.org/docs/current/pgbench.html*